

RLZAP: Relative Lempel-Ziv with Adaptive Pointers^{*}

Anthony J. Cox¹, Andrea Farruggia², Travis Gagie^{3,4},
Simon J. Puglisi^{3,4}, and Jouni Sirén⁵

¹ Illumina Cambridge Ltd, UK

² University of Pisa, Italy

³ Helsinki Institute for Information Technology, Finland

⁴ University of Helsinki, Finland

⁵ Wellcome Trust Sanger Institute, UK

Abstract. Relative Lempel-Ziv (RLZ) is a popular algorithm for compressing databases of genomes from individuals of the same species when fast random access is desired. With Kuruppu et al.’s (SPIRE 2010) original implementation, a reference genome is selected and then the other genomes are greedily parsed into phrases exactly matching substrings of the reference. Deorowicz and Grabowski (*Bioinformatics*, 2011) pointed out that letting each phrase end with a mismatch character usually gives better compression because many of the differences between individuals’ genomes are single-nucleotide substitutions. Ferrada et al. (SPIRE 2014) then pointed out that also using relative pointers and run-length compressing them usually gives even better compression. In this paper we generalize Ferrada et al.’s idea to handle well also short insertions, deletions and multi-character substitutions. We show experimentally that our generalization achieves better compression than Ferrada et al.’s implementation with comparable random-access times.

1 Introduction

Next-generation sequencing technologies can quickly and cheaply yield far more genetic data than can fit into an everyday computer’s memory, so it is important to find ways to compress it while still supporting fast random access. Often the data is highly repetitive and can thus be compressed very well with LZ77 [1], but then random access is slow. For many applications, however, we need store only a database of genomes from individuals of the same species, which are not only highly repetitive collectively but also but also all very similar to each other. Kuruppu, Puglisi and Zobel [2] proposed choosing one of the genomes as a reference and then greedily parsing each of the others into phrases exactly matching substrings of that reference. They called their algorithm Relative Lempel-Ziv (RLZ)

^{*} Supported by the Academy of Finland through grants 258308, 268324, 284598 and 285221. Parts of this work were done during the second author’s visit to the University of Helsinki and during the third author’s visits to Illumina Cambridge Ltd. and the University of A Coruña, Spain.

because it can be viewed as a version of LZ77 that looks for phrase sources only in the reference, which greatly speeds up random access later. (Ziv and Merhav [3] introduced a similar algorithm for estimating the relative entropy of the sources of two sequences.) RLZ is now popular for compressing not only such genomic databases but also other kinds of repetitive datasets; see, e.g., [4,5]. Deorowicz and Grabowski [6] pointed out that letting each phrase end with a mismatch character usually gives better compression on genomic databases because many of the differences between individuals’ genomes are single-nucleotide substitutions, and gave a new implementation with this optimization. Ferrada, Gagic, Gog and Puglisi [7] then pointed out that often the current phrase’s source ends two characters before the next phrase’s source starts, so the distances between the phrases’ starting positions and their sources’ starting positions are the same. They showed that using relative pointers and run-length compressing them usually gives even better compression on genomic databases.

In this paper we generalize Ferrada et al.’s idea to handle well also short insertions, deletions and substitutions. In the Section 2 we review in detail RLZ and Deorowicz and Grabowski’s and Ferrada et al.’s optimizations. We also discuss how RLZ can be used to build relative data structures and why the optimizations that work to better compress genomic databases fail for this application. In Section 3 we explain the design and implementation of RLZ with adaptive pointers (RLZAP): in short, after parsing each phrase, we look ahead several characters to see if we can start a new phrase with a similar relative pointer; if so, we store the intervening characters as mismatch characters and store the new relative pointer encoded as its difference from the previous one. We present our experimental results in Section 4, showing that RLZAP achieves better compression than Ferrada et al.’s implementation with comparable random-access times. Finally, in Section 5 we discuss directions for future work. Our implementation and datasets are available for download from <http://github.com/farruggia/rlzap>.

2 Preliminaries

In this section we discuss the previous work that is the basis and motivation for this paper. We first review in greater detail Kuruppu et al.’s implementation of RLZ and Deorowicz and Grabowski’s and Ferrada et al.’s optimizations. We then quickly summarize the new field of *relative data structures* — which concerns when and how we can use compress a new instance of a data structure, using an instance we already have for a similar dataset — and explain how it uses RLZ and why it needs a generalization of Deorowicz and Grabowski’s and Ferrada et al.’s optimizations.

2.1 RLZ

To compute the RLZ parse of a string $S[1..n]$ with respect to a reference string R using Kuruppu et al.’s implementation, we greedily parse S from left to right

into phrases

$$\begin{aligned}
S[p_1 = 1..p_1 + \ell_1 - 1] \\
S[p_2 = p_1 + \ell_1..p_2 + \ell_2 - 1] \\
\vdots \\
S[p_t = p_{t-1} + \ell_{t-1}..p_t + \ell_t - 1 = n]
\end{aligned}$$

such that each $S[p_i..p_i + \ell_i - 1]$ exactly matches some substring $R[q_i..q_i + \ell_i - 1]$ of R — called the i th phrase’s *source* — for $1 \leq i \leq t$, but $S[p_i..p_i + \ell_i]$ does not exactly match any substring in R for $1 \leq i \leq t - 1$. For simplicity, we assume R contains every distinct character in S , so the parse is well-defined.

Suppose we have constant-time random access to R . To support constant-time random access to S , we store an array $Q[1..t]$ containing the starting positions of the phrases’ sources, and a compressed bitvector $B[1..n]$ with constant query time (see, e.g., [8] for a discussion) and 1s marking the first character of each phrase. Given a position j between 1 and n , we can compute in constant time

$$S[i] = R[Q[B.\text{rank}(j)] + j - B.\text{select}(B.\text{rank}(j))].$$

If there are few phrases then Q is small and B is sparse, so we use little space.

For example, if

$$\begin{aligned}
R &= \text{ACATCATTCGAGGACAGGTATAGCTACAGTTAGAA} \\
S &= \text{ACATGATTCGACGACAGGTACTAGCTACAGTAGAA}
\end{aligned}$$

then we parse S into

$$\text{ACAT, GA, TTCGA, CGA, CAGGTA, CTA, GCTACAGT, AGAA,}$$

and store

$$\begin{aligned}
Q &= 1, 10, 7, 9, 15, 24, 23, 32 \\
B &= 10001010000100100000100100000001000.
\end{aligned}$$

To compute $S[25]$, we compute $B.\text{rank}(25) = 7$ and $B.\text{select}(7) = 24$, which tell us that $S[25]$ is $25 - 24 = 1$ character after the initial character in the 7th phrase. Since $Q[7] = 23$, we look up $S[25] = R[24] = \text{C}$.

2.2 GDC

Deorowicz and Grabowski [6] pointed out that with Kuruppu et al.’s implementation of RLZ, single-character substitutions usually cause two phrase breaks: e.g., in our example $S[1..11] = \text{ACATGATTCGA}$ is split into three phrases, even though the only difference between it and $R[1..11]$ is that $S[5] = \text{G}$ and $R[5] = \text{C}$. They proposed another implementation, called the Genome Differential Compressor (GDC), that lets each phrase end with a mismatch character — as the

original version of LZ77 does — so single-character substitutions usually cause only one phrase break. Since many of the differences between individuals’ DNA are single-nucleotide substitutions, GDC usually compresses genomic databases better than Kuruppu et al.’s implementation.

Specifically, with GDC we parse S from left to right into phrases $S[p_1..p_1 + \ell_1], S[p_2 = p_1 + \ell_1 + 1..p_2 + \ell_2], \dots, S[p_t = p_{t-1} + \ell_{t-1} + 1..p_t + \ell_t = n]$ such that each $S[p_i..p_i + \ell_i - 1]$ exactly matches some substring $R[q_i..q_i + \ell_i - 1]$ of R — again called the i th phrase’s source — for $1 \leq i \leq t$, but $S[p_i..p_i + \ell_i]$ does not exactly match any substring in R , for $1 \leq i \leq t - 1$.

Suppose again that we have constant-time random access to R . To support constant-time random access to S , we store an array $Q[1..t]$ containing the starting positions of the phrases’ sources, an array $M[1..t]$ containing the last character of each phrase, and a compressed bitvector $B[1..n]$ with constant query time and 1s marking the last character of each phrase. Given a position j between 1 and n , we can compute in constant time

$$S[j] = \begin{cases} M[B.\text{rank}(j)] & \text{if } B[j] = 1, \\ R[Q[B.\text{rank}(j) + 1] + j - B.\text{select}(B.\text{rank}(j)) - 1] & \text{otherwise,} \end{cases}$$

assuming $B.\text{select}(0) = 0$.

In our example, we parse S into

ACATG, ATTCGAC, GACAGGTAC, TAGCTACAGT, AGAA,

and store

$Q = 1, 6, 13, 21, 32$

$M = \text{GCCTA}$

$B = 00001000000100000000100000000010001.$

To compute $S[25]$, we compute $B[25] = 0$, $B.\text{rank}(25) = 3$ and $B.\text{select}(3) = 21$, which tell us that $S[25]$ is $25 - 21 - 1 = 3$ characters after the initial character in the 4th phrase. Since $Q[4] = 21$, we look up $S[25] = R[24] = \text{C}$.

2.3 Relative pointers

Ferrada, Gagie, Gog and Puglisi [7] pointed out that after a single-character substitution, the source of the next phrase in GDC’s parse often starts two characters after the end of the source of the current phrase: e.g., in our example the source for $S[1..5] = \text{ACATG}$ is $R[1..4] = \text{ACAT}$ and the source for $S[6..12] = \text{ATTCGAC}$ is $R[6..11] = \text{ATTCGA}$. This means the distances between the phrases’ starting positions and their sources’ starting positions are the same. They proposed an implementation of RLZ that parses S like GDC does but keeps a relative pointer, instead of the explicit pointer, and stores the list of those relative pointers run-length compressed. Since the relative pointers usually do not change after single-nucleotide substitutions, RLZ with relative pointers usually

gives even better compression than GDC on genomic databases. (We note that Deorowicz, Danek and Niemiec [9] recently proposed a new version of GDC, called GDC2, that has improved compression but does not support fast random access.)

Suppose again that we have constant-time random access to R . To support constant-time random access to S , we store the array M of mismatch characters and the bitvector B as with GDC. Instead of storing Q , we build an array $D[1..t]$ containing, for each phrase, the difference $q_i - p_i$ between its source's starting position and its own starting position. We store D run-length compressed: i.e., we partition it into maximal consecutive subsequences of equal values, store an array V containing one copy of the value in each subsequence, and a bitvector $L[1..t]$ with constant query time and 1s marking the first value of each subsequence. Given k between 1 and t , we can compute in constant time

$$D[k] = V[L.\text{rank}(k)].$$

Given a position j between 1 and n , we can compute in constant time

$$S[j] = \begin{cases} M[B.\text{rank}(j)] & \text{if } B[j] = 1, \\ R[D[B.\text{rank}(j) + 1] + j] & \text{otherwise.} \end{cases}$$

In our example, we again parse S into

ACATG, ATTCGAC, GACAGGTAC, TAGCTACAGT, AGAA,

and store

$$M = \text{GCCTA}$$

$$B = 0000100000001000000000100000000010001,$$

but now we store $D = 0, 0, 0, -1, 0$ as $V = 0, -1, 0$ and $L = 10011$ instead of storing Q . To compute $S[25]$, we again compute $B[25] = 0$ and $B.\text{rank}(25) = 3$, which tell us that $S[25]$ is in the 4th phrase. We add 25 to the 4th relative pointer $D[4] = V[L.\text{rank}(4)] = -1$ and obtain 24, so $S[25] = R[24]$.

A single-character insertion or deletion usually causes only a single phrase break in the parse but a new run in D , with the values in the run being one less or one more than the values in the previous run. In our example, the insertion of $S[21] = \text{C}$ causes the value to decrement to -1, and the deletion of $R[26] = \text{T}$ (or, equivalently, of $R[27] = \text{T}$) causes the value to increment to 0 again. In larger examples, where the values of the relative pointers are often a significant fraction of n , it seems wasteful to store a new value uncompressed when it differs only by 1 from the previous value.

For example, suppose R and S are thousands of characters long,

$$R[1783..1817] = \dots \text{ACATCATTTCGAGGACAGGTATAGCTACAGTTAGAA} \dots$$

$$S[2009..2043] = \dots \text{ACATGATTTCGACGACAGGTACTAGCTACAGTAGAA} \dots$$

and GDC still parses $S[2009..2043]$ into the same phrases as before, with their sources in $R[1783..1817]$. The relative pointers for those phrases are $-136, -136, -136, -137, -136$, so we store $-136, -137, -136$ for them in V , which takes at least a couple of dozen bits without further compression.

2.4 Relative data structures

As mentioned in Section 1, the new field of relative data structures concerns when and how we can use compress a new instance of a data structure, using an instance we already have for a similar dataset. Suppose we have a basic FM-index [10] for R — i.e., a rank data structure over the Burrows-Wheeler Transform (BWT) [11] of R , without a suffix-array sample — and we want to use it to build a very compact basic FM-index for S . Since R and S are very similar, it is not surprising that their BWTs are also fairly similar:

BWT(R) = AAGGT\$TTGCCTCCAAATTGAGCAAAGACTAGATGA

BWT(S) = AAGGT\$GTTTCCCGAAAATGAACCTAAGACGGCTAA.

Belazzougui, Gog, Gagie, Manzini and Sirén [12] (see also [13]) showed how we can implement such a relative FM-index for S by choosing a common subsequence of the two BWTs and then storing bitvectors marking the characters not in that common subsequence, and rank data structures over those characters. They also showed how to build a relative suffix-array sample to obtain a fully-functional relative FM-index for S , but reviewing that is beyond the scope of this paper.

An alternative to Belazzougui et al.’s basic approach is to compute the RLZ parse of BWT(S) with respect to BWT(R) and then store the rank for each character just before the beginning of each phrase. We can then answer a rank query BWT(S).rank $_X(j)$ by finding the beginning BWT(S)[p] of the phrase containing BWT(S)[j] and the beginning BWT(R)[q] of that phrase’s source, then computing

$$\text{BWT}(S).\text{rank}_X(p-1) + \text{BWT}(R).\text{rank}_X(q+j-p) - \text{BWT}(R).\text{rank}_X(q-1).$$

Unfortunately, single-character substitutions between R and S usually cause insertions, deletions and multi-character substitutions between BWT(R) and BWT(S), so Deorowicz and Grabowski’s and Ferrada et al.’s optimizations no longer help us, even when the underlying strings are individuals’ genomes. On the other hand, on average those insertions, deletions and multi-character substitutions are fairly few and short [14], so there is still hope that those optimized parsing algorithms can be generalized and applied to make this alternative practical.

Our immediate concern is with a recent implementation of relative suffix trees [15], which uses relative FM-indexes and relatively-compressed longest-common-prefix (LCP) arrays. Deorowicz and Grabowski’s and Ferrada et al.’s optimizations also fail when we try to compress the LCP arrays, and when we use

Kuruppu et al.’s implementation of RLZ the arrays take a substantial fraction of the total space. In our example, however,

$$\text{LCP}(R) = 0,1,1,4,3,1,2,2,3,2,1,2,2,0,3,2,3,1,1,0,2,2,1,1,2,1,2,0,2,3,2,1,2,1,2$$

$$\text{LCP}(S) = 0,1,1,4,3,2,2,1,2,2,2,1,2,0,3,2,1,4,1,3,0,2,3,2,1,1,1,3,0,3,2,3,1,1,1$$

are quite similar: e.g., they have a common subsequence of length 26, almost three quarters of their individual lengths. LCP values tend to grow at least logarithmically with the size of the strings, so good compression becomes more important.

3 Adaptive Pointers

We generalize Ferrada et al.’s optimization to handle short insertions, deletions and substitutions by introducing *adaptive pointers* and by allowing more than one mismatch character at the end of each phrase. An adaptive pointer is represented as the difference from the previous non-adaptive pointer. Henceforth we say a phrase is *adaptive* if its pointer is adaptive, and *explicit* otherwise. In this section we first describe our parsing strategy and then describe how we can support fast random access.

3.1 Parsing

The parsing strategy is a generalization of the Greedy approach for adaptive phrases. The parser first compute the *matching statistics* between input S and reference R : for each suffix $S[i;n]$ of S , a suffix $R[k;m]$ of R with the longest LCP with $S[i]$ is found. Let $\text{MatchPtr}(i)$ be the relative pointer $k - i$ and $\text{MatchLen}(i)$ be the length of the LCP between the two suffixes $S[i;n]$ and $R[k;m]$.

Parsing scans S from left to right, in one pass. Let us assume S has already been parsed up to a position i , and let us assume the most recent explicit phrase starts at position h . The parser first tries to find an adaptive phrase (*adaptive step*); if it fails, looks for an explicit phrase (*explicit step*). Specifically:

1. *adaptive step*: the parser checks, for the current position i if (i) the relative pointer $\text{MatchPtr}(i)$ can be represented as an adaptive pointer, that is, if the differential $\text{MatchPtr}(i) - \text{MatchPtr}(j)$ can be represented as a signed binary integer of at most DeltaBits bits, and (ii) if it is convenient to start a new adaptive phrase instead of representing literals as they are, that is, whether $\text{MatchLen}(i) \cdot \log \sigma > \text{DeltaBits}$. The parser outputs the adaptive phrase and advances $\text{MatchLen}(i)$ positions if both conditions are satisfied; otherwise, it looks for the leftmost position k in range $i + 1$ up to $i + \text{LookAhead}$ where both conditions are satisfied. If it finds such position k , the parser outputs literals $S[i; k - 1]$ and an adaptive phrase; otherwise, it goes to step 2.
2. *explicit step*: in this step the parser goes back to position i and scans forward until it has found a position $k \geq i$ where at least one of these two conditions is satisfied: (i) match length $\text{MatchLen}(i)$ is greater than a parameter ExplicitLen ;

- (ii) the match is followed by an adaptive phrase. It then outputs a literal range $S[i; k - 1]$ and the explicit phrase found.

The purpose of the two conditions on the explicit phrase is to avoid having spurious explicit phrases which are not associated to a meaningfully aligned substrings.

It is important to notice that our data structure logically represents an adaptive/explicit phrase followed by a literal run as a single phrase: for example, an adaptive phrase of length 5 followed by a literal sequence **GAT** is represented as an adaptive phrase of length 8 with the last 3 symbols represented as literals.

3.2 Representation

In order to support fast random access to S , we deploy several data structures, which can be grouped into two sets with different purposes:

1. **Storing the parsing:** a set of data structures mapping any position i to some useful information about the phrase P_i containing $S[i]$, that is: (i) the position $\text{Start}(i)$ of the first symbol in P_i ; (ii) P_i 's length $\text{Len}(i)$; (iii) its relative pointer $\text{Rel}(i)$; (iv) the number of phrases $\text{Prev}(i)$ preceding P_i in the parsing, and (v) the number of explicit phrases $\text{Abs}(i) \geq \text{Prev}(i)$ preceding P_i .
2. **Storing the literals:** a set of data structures which, given a position i and the information about phrase P_i , tells whether $S[i]$ is a literal in the parsing and, if this is the case, returns $S[i]$.

Here we provide a detailed illustration of these data structures.

Storing the parsing. The parsing is represented by storing two bitvectors. The first bitvector **P** has $|S|$ entries, marking with a 1 characters in S at the beginning of a new phrase in the parsing. The second bitvector **E** has m entries, one for every phrases in the parsing, and marks every explicit phrase in the parsing with a 1, otherwise 0. A rank/select datastructure is built on top of **P**, and a rank datastructure on top of **E**. In this way, given i we can efficiently compute the phrase index $\text{Prev}(i)$ as $\text{P.rank}(i)$, the explicit phrase index $\text{Abs}(i)$ as $\text{E.rank}(p_i)$ and the phrase beginning $\text{Start}(i)$ as $\text{P.select}(p_i)$.

Experimentally, bitvector **P** is sparse, while **E** is usually dense. Bitvector **P** can be represented with any efficient implementation for sparse bitvectors; our implementation, detailed in Section 4, employs the Elias-Fano based **SDarrays** datastructure of Okanohara and Sadakane [16], which requires $m \log \frac{|S|}{m} + O(m)$ bits and supports rank in $O(\log \frac{|S|}{m})$ time and select in constant time. Bitvector **E** is represented plainly, taking m bits, with any $o(m)$ -space $O(1)$ -time rank implementation on top of it ([16,17]). In particular, it is interesting to notice that only one **rank** query is needed for extracting an unbounded number of consecutive symbols from **E**, since each starting position of consecutive phrases can be accessed with a single **select** query, which has very efficient implementations on sparse bitvectors.

Both explicit and relative pointers are stored using minimal binary codes in tables A and R , respectively. These integers are not compressed using statistical encoding because this would prevent efficient random access to the sequence. Each explicit and relative pointer takes thus $\lceil \log n \rceil$ and $\lceil \log (\text{LookAhead}) \rceil + 1$ bits of space, respectively. To compute $\text{Rel}(i)$, we first check if the phrase is explicit by checking if $S[\text{Abs}(i)]$ is set to one; if it is, then $\text{Rel}(i) = A[\text{Abs}(i)]$, otherwise it is $\text{Rel}(i) = A[\text{Abs}(i)] + R[\text{Prev}(i) - \text{Abs}(i)]$.

Storing literals. Literals are extracted as follows. Let us assume we are interested in accessing $S[i]$, which is contained in phrase P_j . First, it is determined whether $S[i]$ is a literal or not. Since literals in a phrase are grouped at the end of the phrase itself, it is sufficient to store, for every phrase P_k in the parsing, the number of literals $\text{Lits}(k)$ at its end. Thus, knowing the starting position $\text{Start}(j)$ and length $\text{Len}(j)$ of phrase P_j , symbol $S[i]$ is a literal if and only if $i > \text{Start}(j) + \text{Len}(j) - \text{Lits}(j)$.

All literals are stored in a table L , where $L[k]$ is the k -th literal found by scanning the parsing from left to right. How we represent L depends on the kind of data we are dealing with. In our experiments, described in Section 4, we consider differentially-encoded LCP arrays and DNA. For DLCP values, L simply stores all values using minimal binary codes. For DNA values, a more refined implementation (which we describe in a later paragraph) is needed to use less than 3 bits on average for each symbol. So, in order to display the literal $S[i]$, we need a way to compute its index in L , which is equal to $\text{Start}(j) - \text{Len}(j) - \text{Lits}(k)$ plus the prefix sum $\sum_{k=1}^{j-1} \text{Lits}(k)$. In the following paragraph we detail two solutions for efficiently storing $\text{Lits}(k)$ values and computing prefix sums.

Storing literal counts. Here we detail a simple and fast data structure for storing $\text{Lits}(-)$ values and for computing prefix sums on them. The basic idea is to store $\text{Lits}(-)$ values explicitly, and accelerate prefix sums by storing the prefix sum of some regularly sampled positions. To provide fast random access, the maximum number of literals in a phrase is limited to $2^{\text{MaxLit}} - 1$, where MaxLit is a parameter chosen at construction time. Every value $\text{Lits}(-)$ is thus collected in a table L , stored using MaxLit bits each. Since each phrase cannot have more than $2^{\text{MaxLit}} - 1$ literals, we split each run of more than $2^{\text{MaxLit}} - 1$ literals into the minimal number of phrases which do meet the limit. In order to speed-up the prefix sum computation on L , we sample one every SampleInt positions and store prefix sums of sampled positions into a table Prefix . To accelerate further prefix sum computation, we employ a 256-entries table Byte_Σ which maps any sequence of $8/\text{MaxLit}$ elements into their sum. Here, we constrain MaxLit as a power of two not greater than 8 (that is, either 1, 2, 4 or 8) and SampleInt as a multiple of $8/\text{MaxLit}$. In this way we can compute the prefix sum by just one look-up into Prefix and at most $\frac{\text{SampleInt}}{8/\text{MaxLit}}$ queries into Byte_Σ . Using Byte_Σ is faster than summing elements in L because it replaces costly bitshift operations with efficient byte-accesses to L . This is because $8/\text{MaxLit}$ elements of L fit into one byte; moreover, those bytes are aligned to byte-boundaries because SampleInt is a multiple of $8/\text{MaxLit}$, which in turn implies that the sampling interval spans entire bytes of L .

Storing DNA literals. Every literal is collected into a table J , where each element is represented using a fixed number of bits. For the DNA sequences we consider in our experiments, this would imply using 3 bits, since the alphabet is $\{A, C, G, T, N\}$. However, since symbols N occur less often than the others, it is more convenient to handle those as exceptions, so other literals can be stored in just 2 bits. In particular, every N in table J is stored as one of the other four symbols in the alphabet (say, A) and a bit-vector **Exc** marks every position in J which corresponds to an N . Experimentally, bitvector **Exc** is sparse and the 1 are usually clustered together into a few regions. In order to reduce the space needed to store **Exc**, we designed a simple bit-vector implementation to exploit this fact. In our design, **Exc** is divided into equal-sized chunks of length C . A bitvector **Chunk** marks those chunks which contain at least one bit set to 1. Marked chunks of **Exc** are collected into a vector V . Because of the clustering property we just mentioned, most of the chunks are not marked, but marked chunks are locally dense. Because of this, bitvector **Chunk** is implemented using a sparse representation, while each chunk employs a dense representation. Good experimental values for C are around 16 – 32 bits, so each chunk is represented with a fixed-width integer. In order to check whether a position i is marked in **Exc**, we first check if chunk $c = \lfloor i/C \rfloor$ is marked in **Chunk**. If it is marked, we compute **Chunk.rank**(c) to get the index of the marked chunk in V .

4 Experiments

We implemented RLZAP in C++11 with bitvectors from Gog et al.’s `sdsl` library (<https://github.com/simongog/sdsl-lite>), and compiled it with `gcc` version 4.8.4 with flags `-O3, -march=native, -ffast-math, -funroll-loops` and `-DNDEBUG`. We performed our experiments on a computer with a 6-core Intel Xeon X5670 clocked at 2.93GHz, 40GiB of DDR3 ram clocked at 1333MHz and running Ubuntu 14.04. As noted in Section 1, our code is available at <http://github.com/farruggia/rlzap>.

We performed our experiments on the following four datasets:

- **Cere**: the genomes of 39 strains of the *Saccharomyces cerevisiae* yeast;
- **E. Coli**: the genomes of 33 strains of the *Escherichia coli* bacteria;
- **Para**: the genomes of 36 strains of the *Saccharomyces paradoxus* yeast;
- **DLCP**: differentially-encoded LCP arrays for three human genomes, with 32-bit entries.

These files are available from <http://acube.di.unipi.it/rlzap-dataset>.

For each dataset we chose the file (i.e., the single genome or DLCP array) with the lexicographically largest name to be the *reference*, and made the concatenation of the other files the *target*. We then compressed the target against the reference with Ferrada et al.’s optimization of RLZ — which reflects the current state of the art, as explained in Section 1 — and with RLZAP. For the DNA files (i.e., **Cere**, **E. Coli** and **Para**) we used `LookAhead = 32`, `MinLen = 32` and `DeltaBits = 2`, while for **DLCP** we used `LookAhead = 8`, `MinLen = 4` and

Table 1. Compression achieved by RLZ and RLZAP. For each dataset we report in MiB (2^{20} bytes) the size of the reference and the size of the target uncompressed and compressed with each method.

Dataset	Reference size (MiB)	Target size (MiB)	Compressed Target Size (MiB)	
			RLZ	RLZAP
Cere	12.0	451	9.16	7.61
E. Coli	4.8	152	30.47	21.51
Para	11.3	398	15.57	10.49
DLCP	11,582	23,392	1,745.33	1,173.81

Table 2. Extraction times per character from RLZ- and RLZAP-compressed targets. For each file in each target, we compute the mean extraction time for $2^{24}/\ell$ pseudo-randomly chosen substrings; take the mean of these means.

Dataset	Algorithm	Mean extraction time per character (ns)					
		1	4	16	64	256	1024
Cere	RLZ	234	59	16.4	4.4	1.47	0.55
	RLZAP	274	70	19.5	5.7	2.34	1.26
E. Coli	RLZ	225	62	20.1	7.7	4.34	3.34
	RLZAP	322	91	31.3	15.3	10.78	9.47
Para	RLZ	235	59	17.2	5.2	2.23	1.03
	RLZAP	284	74	21.2	6.9	3.09	2.26
DLCP	RLZ	756	238	61.5	20.5	9.00	6.00
	RLZAP	826	212	57.5	19.0	8.00	4.50

DeltaBits = 4. We chose these parameters during a calibration step performed on a different dataset, which we will describe in the full version of this paper.

Table 1 shows the compression achieved by RLZ and RLZAP. (We note that, since the DNA datasets are each over an alphabet of $\{A, C, G, T, N\}$ and Ns are rare, the targets for those datasets can be compressed to about a quarter of their size even with only, e.g., Huffman coding.) Notice RLZAP consistently achieves better compression than RLZ, with its space usage ranging from about 17% less for Cere to about 32% less for DLCP.

Table 2 shows extraction times for RLZ- and RLZAP-compressed targets. RLZAP is noticeably slower than RLZ for DNA, while it is slightly faster for the DLCP dataset when at least four characters are extracted. We believe RLZAP outperforms RLZ on the DLCP because its parsing is generally more cache-friendly: our measurements indicate that on this dataset RLZAP causes about 36% fewer L2 and L3 cache misses than RLZ. Even for DNA, RLZAP is still fast

in absolute terms, taking just tens of nanoseconds per character when extracting at least four characters.

On DNA files, RLZAP achieves better compression at the cost of slightly longer extraction times. On differentially-encoded LCP arrays, RLZAP outperforms RLZ in all regards, except for a slight slowdown when extraction substrings of length less than 4. That is, RLZAP is competitive with the state of the art even for compressing DNA and, as we hoped, advances it for relative data structures. Our next step will be to integrate it into the implementation of relative suffix trees mentioned in Subsection 2.4.

5 Future Work

In the near future we plan to perform more experiments to tune RLZAP and discover its limitations. For example, we will test it on the balanced-parentheses representations of suffix trees' shapes, which are an alternative to LCP arrays, and on the BWTs in relative FM-indexes. We also plan to investigate how to minimize the bit-complexity of our parsing — i.e., how to choose the phrases and sources so as to minimize the number of bits in our representation — building on the results by Farruggia, Ferragina and Venturini [18,19] about minimizing the bit-complexity of LZ77.

RLZAP can be viewed as a bounded-lookahead greedy heuristic for computing a glocal alignment [20] or S against R . Such an alignment allows for genetic recombination events, in which potentially large sections of DNA are rearranged. We note that standard heuristics for speeding up edit-distance computation and global alignment do not work here, because even a low-cost path through the dynamic programming matrix can occasionally jump arbitrarily far from the diagonal. RLZAP runs in linear time, which is attractive, but it may produce a suboptimal alignment — i.e., it is not an admissible heuristic. In the longer term, we are interested in finding practical admissible heuristics.

For example, if a long enough substring of S aligns well enough against a particular substring of R and badly enough against any other substring or small collection of substrings of R (which we can check with LCP queries), then any optimal alignment of S against R should align most of that subalignment. This observation should help us find an optimal alignment when the RLZ parse of S with respect to R is small but, e.g., there are few or no long approximate repetitions within R , so the LZ77 parse of R is fairly large.

Apart from the direct biological interest of computing optimal or nearly optimal glocal alignments, they can also help us design more data structures. For example, consider the problem of representing the mapping between orthologous genes in several species' genomes; see, e.g., [21]. Given two genomes' indices and the position of a base-pair in one of those genomes, we would like to return quickly the positions of all corresponding base-pairs in the other genome. Only a few base-pairs correspond to two base-pairs in another genome and, ignoring those, this problem reduces to representing compressed permutations. A feature of these permutations is that base-pairs tend to be mapped in blocks, possibly

with some slight reordering within each block. We can extract this block structure by computing a glocal alignment, either between the genomes or between the permutation and its inverse.

References

1. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory* **23** (1977) 337–343
2. Kuruppu, S., Puglisi, S.J., Zobel, J.: Relative Lempel-Ziv compression of genomes for large-scale storage and retrieval. In: *Proc. SPIRE*. (2010) 201–206
3. Ziv, J., Merhav, N.: A measure of relative entropy between individual sequences with application to universal classification. *IEEE Trans. on Inf. Theory* **39** (1993) 1270–1279
4. Hoobin, C., Puglisi, S.J., Zobel, J.: Sample selection for dictionary-based corpus compression. In: *Proc. SIGIR*. (2011) 1137–1138
5. Hoobin, C., Puglisi, S.J., Zobel, J.: Relative Lempel-Ziv factorization for efficient storage and retrieval of web collections. *Proc. VLDB* **5** (2011) 265–273
6. Deorowicz, S., Grabowski, S.: Robust relative compression of genomes with random access. *Bioinformatics* **27** (2011) 2979–2986
7. Ferrada, H., Gagie, T., Gog, S., Puglisi, S.J.: Relative Lempel-Ziv with constant-time random access. In: *Proc. SPIRE*. (2014) 13–17
8. Kärkkäinen, J., Kempa, D., Puglisi, S.J.: Hybrid compression of bitvectors for the FM-index. In: *Proc. DCC*. (2014) 302–311
9. Deorowicz, S., Danek, A., Niemiec, M.: GDC2: Compression of large collections of genomes. *Scientific Reports* **5** (2015)
10. Ferragina, P., Manzini, G.: Indexing compressed text. *Journal of the ACM* **52** (2005) 552–581
11. Burrows, M., Wheeler, D.J.: A block sorting lossless data compression algorithm. Technical Report 124, Digital Equipment Corporation (1994)
12. Belazzougui, D., Gagie, T., Gog, S., Manzini, G., Sirén, J.: Relative FM-indexes. In: *Proc. SPIRE*. (2014) 52–64
13. Boucher, C., Bowe, A., Gagie, T., Manzini, G., Sirén, J.: Relative select. In: *Proc. SPIRE*. (2015) 149–155
14. Léonard, M., Mouchard, L., Salson, M.: On the number of elements to reorder when updating a suffix array. *J. Discrete Algorithms* **11** (2012) 87–99
15. Gagie, T., Navarro, G., Puglisi, S.J., Sirén, J.: Relative compressed suffix trees. Technical Report 1508.02550, arxiv.org (2015)
16. Okanohara, D., Sadakane, K.: Practical entropy-compressed rank/select dictionary. In: *Proc. ALENEX*. (2007)
17. Raman, R., Raman, V., Satti, S.R.: Succinct indexable dictionaries with applications to encoding k -ary trees, prefix sums and multisets. *ACM Trans. Algorithms* **3** (2007)
18. Farruggia, A., Ferragina, P., Venturini, R.: Bicriteria data compression. In: *Proc. SODA*. (2014) 1582–1595
19. Farruggia, A., Ferragina, P., Venturini, R.: Bicriteria data compression: Efficient and usable. In: *Proc. ESA*. (2014) 406–417
20. Brudno, M., Malde, S., Poliakov, A., Do, C.B., Couronne, O., Dubchak, I., Batzoglou, S.: Glocal alignment: finding rearrangements during alignment. In: *Proc. ISMB*. (2003) 54–62
21. Kubincová, P.: Mapping between Genomes. Bachelor thesis, Comenius University, Slovakia (2014) Supervised by Broňa Brejová.